14 Message Passing Interface

MPI = Message Passing Interface

Caractéristiques de MPI :

- **Bibliothèque** — pour échanges de messages
- Plusieurs langages — Fortran, C/C++, Java, Perl, Python, R, Ruby, ...
- Devenu un standard *de facto*
- Relativement complexe :
  - MPI 1.0 : 129 constantes et fonctions
  - MPI 2.0 : 249 constantes et fonctions
  - MPI 3.2 : 396 constantes et fonctions

Note : La première partie décrit essentiellement MPI 1.0.

14.1 Le modèle de programmation MPI

- Création **statique** des processus : un nombre fixe de processus est créé au lancement du programme

  Note : Il s'agit processus *«poids lourd» (avec espace mémoire privé)*
  ⇒ Programme!

- Modèle **SPMD** = *Single Program*, *Multiple Data* (voir page suivante)

- La façon dont les processus sont créés n'est pas décrite dans le standard

- Les processus communiquent en utilisant diverses fonctions :
  - **Communications point-à-point** — entre deux (2) processus
  - **Communications collectives** — entre plusieurs processus

Single Program, Multiple Data (SPMD)

This is the most common way to organize a parallel program, especially on MIMD computers. The idea is that a single program is written and loaded onto each node of a parallel computer. Each copy of the single program runs independently (aside from coordination events), so the instruction streams executed on each node can be completely different. The specific path through the code is in part selected by the node ID.

Source : «*Patterns for parallel programming*», Mattson, Sanders & Massingil, 2005.

SPMD vs. SIMD

In **SPMD***, multiple autonomous processors simultaneously execute the same program at independent points, rather than in the lockstep that* **SIMD** *imposes on different data. With SPMD, tasks can be executed on general purpose CPUs [...].*

Source : `http://en.wikipedia.org/wiki/SPMD`

14.2 Éléments de base de MPI

Les six (6) fonctions de base de MPI — **communications point-à-point** :

- `MPI_Init` : Initialise l'utilisation de MPI
- `MPI_Finalize` : Termine l'utilisation de MPI

- `MPI_Comm_size` : Nombre de processus impliqués
- `MPI_Comm_rank` : Numéro d'identification du processus

- `MPI_Send` : Envoie un message
- `MPI_Recv` : Reçoit un message

Toutes les communications sont **relatives à un communicateur**

Communicateur ≈ espace de communication dans lequel participent un certain nombre de processus

Communicateur de base = `MPI_COMM_WORLD` = tous les processus

Note : Source des descriptions détaillées (synopsis et descriptions des paramètres) des fonctions dans les pages qui suivent :

`http://www.mpich.org/static/docs/v3.2/`

MPI_Init

Initialize the MPI execution environment

Synopsis

```
int MPI_Init( int *argc, char ***argv )
```

Input Parameters

argc Pointer to the number of arguments

argv Pointer to the argument vector

MPI_Finalize

Terminates MPI execution environment

Synopsis

```
int MPI_Finalize( void )
```

MPI_Comm_size

Determines the size of the group associated with a communicator

Synopsis

```
int MPI_Comm_size( MPI_Comm comm, int *size )
```

Input Parameters

comm communicator (handle)

Output Parameters

size number of processes in the group of comm (integer)

MPI_Comm_rank

Determines the rank of the calling process in the communicator

Synopsis

```
int MPI_Comm_rank( MPI_Comm comm, int *rank )
```

Input Parameters

comm communicator (handle)

Output Parameters

rank rank of the calling process in the group of comm (integer)

MPI_Send

Performs a blocking send

Synopsis

```
int MPI_Send(const void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

Input Parameters

buf	initial address of send buffer (choice)
count	number of elements in send buffer (nonnegative integer)
datatype	datatype of each send buffer element (handle)
dest	rank of destination (integer)
tag	message tag (integer)
comm	communicator (handle)

Note : Pour envoyer simplement un **signal** (de synchronisation), on peut écrire :

```
MPI_Send( NULL, 0, MPI_BYTE, ...)
```

Note : *MPI functions sometimes use arguments with a choice (or union) data type. Distinct calls to the same routine may pass by reference actual arguments of different types.*

MPI_Recv

Blocking receive for a message

Synopsis

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
             int tag, MPI_Comm comm, MPI_Status *status)
```

Input Parameters

count — maximum number of elements in receive buffer (integer)

datatype — datatype of each receive buffer element (handle)

source — rank of source (integer)

tag — message tag (integer)

comm — communicator (handle)

Output Parameters

buf — initial address of receive buffer (choice)

status — status object (Status)

Note : On peut aussi écrire ce qui suit, pour ignorer les informations de statut retournées par l'appel :

```
MPI_Recv( ..., MPI_STATUS_IGNORE )
```

Exemple = Impression du numéro de processus :

```
int main( int argc, char *argv[] )
{
  MPI_Init( &argc, &argv );
  MPI_Comm_size( MPI_COMM_WORLD, &count );
  MPI_Comm_rank( MPI_COMM_WORLD, &myid );

  printf( "Processus␣%d␣parmi␣%d\n", myid, count );

  MPI_Finalize();
}
```

Exemple = Stratégie SPMD pour une approche coordonnateur/travailleurs :

```
int main( int argc, char *argv[] )
{
  MPI_Init(&argc, &argv);
  ...
  MPI_Comm_rank(MPI_COMM_WORLD, &mon_id);

  if (mon_id == 0) {
    coordonnateur(); // Instance (1) du programme => coordonnateur.
  } else {
    travailleur();   // Instances (multiples) du programme => travailleurs.
  }
  ...
  MPI_Finalize();
}
```

14.2.1 Les différents langages supportant MPI

MPI est **spécifié** de façon indépendante de tout langage de programmation

Différentes interfaces (*bindings*) ont été développées pour différents langages :

- Fortran (encore très utilisé en programmation scientifique)
- C
- Java
- Etc.

Convention en C pour les identificateurs et interfaces des opérations :

- "`MPI_`" ajouté devant les identificateurs :

  `MPI_Init`

  `MPI_COMM_WORLD`

  `MPI_Comm_size`

- Constantes : en `MAJUSCULES`

  `MPI_COMM_WORLD`

  `MPI_CHAR`

  `MPI_INT`

- Noms d'opérations : première lettre (après `MPI_`) en Majuscule

  `MPI_Init`

  `MPI_Send`

  `MPI_Get_count`

- Résultats retournés par l'intermédiaire de pointeurs vers des variables

- Types de base pré-définis : `MPI_CHAR`, `MPI_INT`, `MPI_FLOAT`, etc.

14.2.2 Déterminisme

Ordre de réception des messages :

- Si un processus `A` envoie un message `m1`, *puis* un message `m2` à un autre processus `C`, alors `m1` sera reçu et obtenu par `C` *avant* `m2`

- Si deux processus `A` et `B` envoient un message à un autre processus `C`, l'ordre d'arrivée à `C` des messages n'est pas déterminé

14.2.3 Communication sans canaux explicites

Les communications point-à-point *ne se font pas* par l'intermédiaire de canaux. Elles se font directement d'un processus à un autre

Mais ... un canal peut être **simulé** en spécifiant l'enveloppe d'un message :

- La source du message (numéro de processus ou `MPI_ANY_SOURCE`)
- Une étiquette (*tag*) associée au message (entier ou `MPI_ANY_TAG`)
- Un contexte (un communicateur : voir Section 14.5)

14.3 Opérations collectives de communication

En théorie : tout pourrait se faire avec `MPI_Send`/`Recv` — $\approx$ «`goto`»

En pratique : les opérations *collectives de communication* — qui impliquent plusieurs processus — peuvent être mises en oeuvre de façon plus efficace par la bibliothèque que par le programmeur

Principales opérations de communication collective

(voir figure plus loin, tirée de `http://www.mcs.anl.gov/~itf/dbpp/text/node97.html`)

- Barrière de synchronisation
- Diffusion d'une donnée spécifique aux autres processus
- Distribution d'un ensemble de données à l'ensemble des processus
- Collecte des données provenant des divers processus
- Application d'une opération de réduction (somme, max, etc.)

14.3.1 Barrière

Opération relative à un communicateur (i.e., un groupe de processus) = assure qu'aucun processus du groupe (`comm`) ne poursuit tant que tous les processus ne sont pas arrivés à la barrière :

```
MPI_Barrier( comm );
```

14.3.2 Déplacement de données

Trois principales opérations permettent à un groupe de processus d'interagir avec un processus *racine* pour déplacer, de façon collective, des données

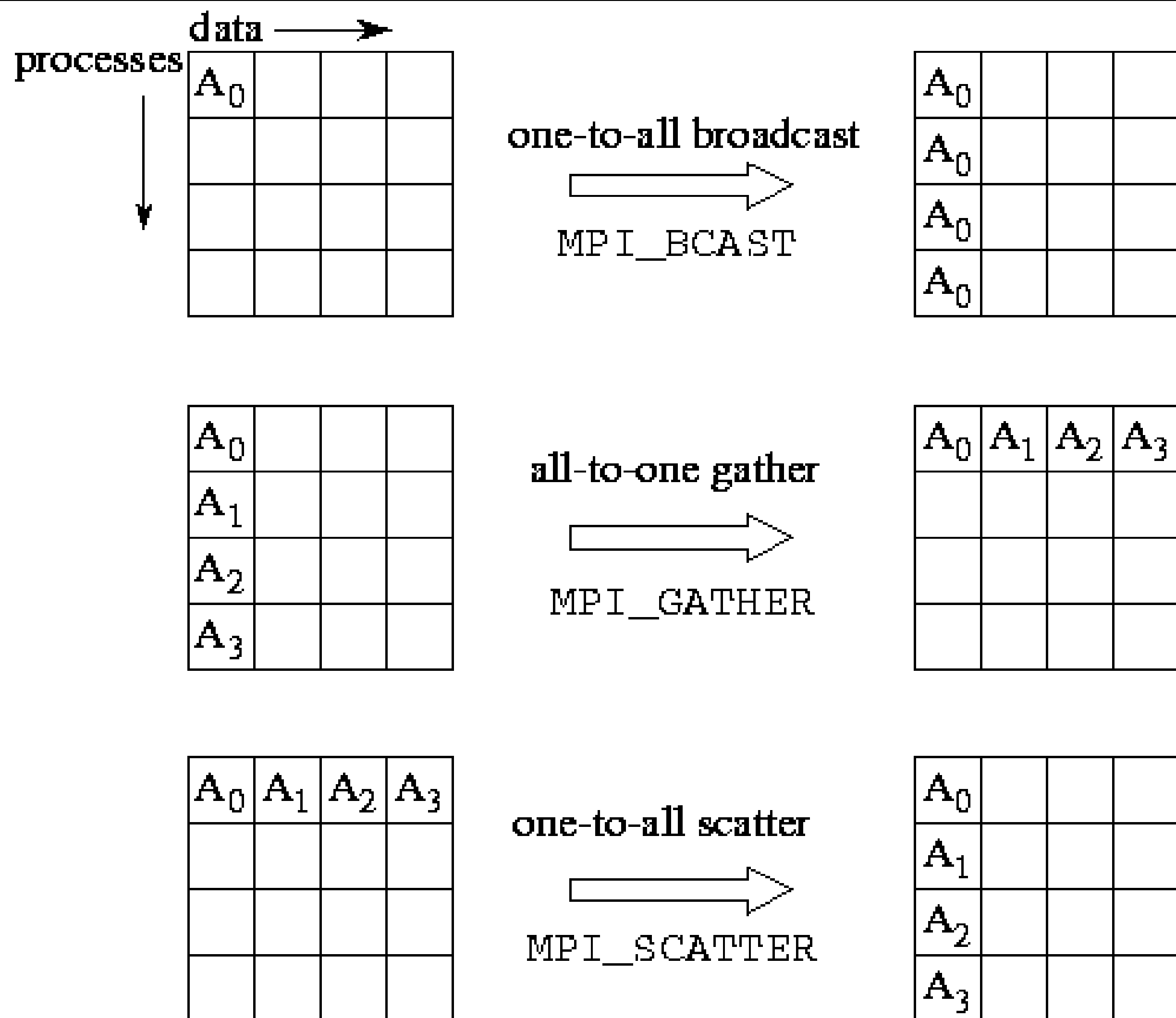


Figure 8.3: *MPI collective data movement functions, illustrated for a group of 4 processes. In each set of 16 boxes, each row represents data locations in a different process. Thus, in the one-to-all broadcast, the data* A_0 *is initially located just in process 0; after the call, it is replicated in all processes. In each case, both* `incnt` *and* `outcnt` *are 1, meaning that each message comprises a single data element.*

1. `MPI_Bcast( tamp, nb_elems, t_elem, racine, comm );`

Envoie `nb_elems` (de type `t_elem`) provenant de `tamp` du processus `racine` vers chacun des autres processus.

⇒ Communication «un vers plusieurs»

MPI_Bcast

Broadcasts a message from the process with rank "root" to all other processes of the communicator

Synopsis

```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root,
               MPI_Comm comm )
```

Input/Output Parameters

buffer   starting address of buffer (choice)

Input Parameters

count   number of entries in buffer (integer)
datatype data type of buffer (handle)
root   rank of broadcast root (integer)
comm   communicator (handle)

```
2. MPI_Scatter( tamp_in, nb_elems_in, t_elem_in,
                tamp_out, nb_elems_out, t_elem_out,
                racine, comm );
```

Inverse de `MPI_Gather` = `racine` envoie la `i`ème portion des données provenant de `tamp_in` au processus `i`

⇒ Communication «un vers plusieurs»

Différences entre `MPI_Bcast` et `MPI_Scatter` :

- `MPI_Bcast` : *la même valeur* est envoyée à chaque processus
- `MPI_Scatter` : des morceaux *différents* sont envoyés à chaque processus

MPI_Scatter

Sends data from one process to all other processes in a communicator

Synopsis

```
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
                void *recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm)
```

Input Parameters

sendbuf	address of send buffer (choice, significant only at root)
sendcount	number of elements sent to each process (integer, significant only at root)
sendtype	data type of send buffer elements (significant only at root) (handle)
recvcount	number of elements in receive buffer (integer)
recvtype	data type of receive buffer elements (handle)
root	rank of sending process (integer)
comm	communicator (handle)

Output Parameters

recvbuf	address of receive buffer (choice)

```
3. MPI_Gather( tamp_in, nb_elems_in, t_elem_in,
               tamp_out, nb_elems_out, t_elem_out,
               racine, comm );
```

Le processus `racine` reçoit dans `tamp_out` les copies des données contenues dans `tamp_in` de chaque processus. Les items du processus `i` sont mis devant ceux de `i+1`.

⇒ Communication «plusieurs vers un»

Note : `tamp_out` est modifié seulement sur `racine` et sa taille doit être P fois la taille de `tamp_in` — où P est le nombre de processus de `comm`.

MPI_Gather

Gathers together values from a group of processes

Synopsis

```
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype,
               void *recvbuf, int recvcount, MPI_Datatype recvtype,
               int root, MPI_Comm comm)
```

Input Parameters

sendbuf   starting address of send buffer (choice)

sendcount number of elements in send buffer (integer)

sendtype data type of send buffer elements (handle)

recvcount number of elements for any single receive (integer, significant only at root)

recvtype data type of recv buffer elements (significant only at root) (handle)

root   rank of receiving process (integer)

comm   communicator (handle)

Output Parameters

recvbuf   address of receive buffer (choice, significant only at root)

14.3.3 Opérations de réduction

Une opération de *réduction* permet de *combiner* (avec une opération binaire) les valeurs contenus dans les tampons des différents processus.

Deux formes de base :

- `MPI_Reduce` : résultat seulement dans le tampon du processus `racine`
- `MPI_Allreduce` : résultat dans le tampon *de chaque processus*

Exemple : Quatre (4) processus avec des tampons de deux éléments

Processus	inbuf		outbuf	
0	2	4	?	?
1	5	7	?	?
2	0	3	?	?
3	6	2	?	?

```
MPI_Reduce( inbuf, outbuf, 2, MPI_INT, MPI_SUM, 3, MPI_COMM_WORLD );
```

Processus	inbuf		outbuf	
0	2	4	?	?
1	5	7	?	?
2	0	3	?	?
3	6	2	13	16

```
MPI_Allreduce( inbuf, outbuf, 2, MPI_INT, MPI_MIN, MPI_COMM_WORLD );
```

Processus	inbuf		outbuf	
0	2	4	0	2
1	5	7	0	2
2	0	3	0	2
3	6	2	0	2

MPI_Reduce

Reduces values on all processes to a single value

Synopsis

```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
               MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm)
```

Input Parameters

sendbuf	address of send buffer (choice)
count	number of elements in send buffer (integer)
datatype	data type of elements of send buffer (handle)
op	reduce operation (handle)
root	rank of root process (integer)
comm	communicator (handle)

Output Parameters

recvbuf	address of receive buffer (choice, significant only at root)

MPI_Allreduce

Combines values from all processes and distributes the result back to all processes

Synopsis

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

Input Parameters

sendbuf	starting address of send buffer (choice)
count	number of elements in send buffer (integer)
datatype	data type of elements of send buffer (handle)
op	operation (handle)
comm	communicator (handle)

Output Parameters

recvbuf	starting address of receive buffer (choice)

14.4 Communications asynchrones

Diverses opérations de communication **asynchrone** sont disponibles :

- `MPI_Isend` : amorce l'envoi d'un message et termine immédiatement en retournant un objet `MPI_Request`

  Le tampon ne peut pas être réutilisé tant que la complétion de l'envoi n'a pas été confirmée, avec `MPI_Wait` (appliqué à l'objet `MPI_Request` reçu)

- `MPI_Irecv` : amorce la réception d'un message et termine immédiatement en retournant un objet `MPI_Request`

  On peut vérifier que la réception a bien été effectuée et complétée avec `MPI_Wait` (appliqué à l'objet `MPI_Request` reçu)

Exemple :

```
MPI_Request request;
MPI_Status status;
...
MPI_Isend( &buf, 1, MPI_INT, dest, 0, MPI_COMM_WORLD, &reques

...
... // On fait autre chose en attendant...
... //     mais sans modifier buf!
...

MPI_Wait( &request, &status );  // Bloque si pas complét.
```

Question : À quoi cela peut-il bien servir?

Question : À quoi cela peut-il bien servir?

Exemple : on amorce la réception d'un message et, pendant qu'elle se complète, on effectue divers autres calculs qui ne dépendent pas des données en attente.

$\Rightarrow$

Permet de superposer (*overlap*) communications et calculs

- `MPI_Probe` : bloque jusqu'à ce qu'un message ayant l'enveloppe (source et *tag*) spécifiée ait été reçu. Ne retire pas (ne «reçoit» pas) le message — retourne simplement le `MPI_status`

- `MPI_Iprobe` : comme `MPI_Probe`, mais de façon immédiate, donc sans bloquer sauf lorsque `MPI_Wait` est appelé sur l'objet `MPI_Request`

- `MPI_Count` : retourne la longueur d'un message identifié avec `MPI_Probe` ou `MPI_Iprobe` via son `MPI_Status`

Exemple = Réception d'un message de longueur variable

```
MPI_Status status;

// On bloque jusqu'à ce qu'un message approprié arrive...
MPI_Probe( MPI_ANY_SOURCE, MPI_ANY_TAG, comm, &status );

// Un message est arrivé: on le traite.
source = status.MPI_SOURCE;
tag = status.MPI_TAG;
error = status.MPI_ERROR; // Code d'erreur le cas échéant.

MPI_Get_count( status, MPI_INT, &count );     // Taille = ?
buf = (int *) malloc(count * sizeof(int)); // Allocation.

MPI_Recv( buf, count, MPI_INT, source, 0, comm, &status );
```

14.5 Modularité et communicateurs

Modularité supportée à l'aide des **communicateurs**

Un communicateur permet de définir un espace **limité** de communication, *privé* à un groupe de processus — donc un **sous-espace**

Quatre (4) principales fonctions pour définir des communicateurs :

1. `MPI_Comm_dup` : Duplique un communicateur existant ⇒ même groupe de processus, mais dans un **contexte distinct**

2. `MPI_Comm_split` : Crée un nouveau communicateur en sélectionnant un certain nombre de processus à l'intérieur d'un groupe existant

3. `MPI_Intercomm_create` : Crée un communicateur qui relie des processus provenant de deux groupes indépendants

4. `MPI_Comm_free` : Détruit un communicateur

14.5.1 Création de communicateurs

Les étiquettes de messages (*tags*) permettent de distinguer entre différentes communications à l'intérieur d'un certain contexte... mais ne permettent pas créer un contexte indépendant, d'où le rôle des communicateurs

Exemple = appel d'une procédure de bibliothèque (parallèle) :

```
MPI_Comm_dup(comm, &newcomm, &ierr);

transpose(A, newcomm);    // Procedure appelée avec
                          // un contexte distinct et indépendant
MPI_Comm_free(&newcomm, &ierr);
```

Les communications faites durant l'exécution de `transpose` se font dans un contexte indépendant du contexte initial d'appel, assurant qu'il n'y aura pas d'interactions indésirées entre le contexte initial et celui de la bibliothèque

14.5.2 Partitionnement de communicateurs et processus

Dans certains langages, il est possible de créer de façon dynamique de nouveaux processus, de nouvelles tâches, *mais pas* en MPI 1.0!

Solution style MPI 1.0 :

- Les processus "*se transforment*" en nouvelles tâches
- Ces processus transformés utilisent des *communicateurs indépendants*, pour éviter les interactions indésirées avec les autres tâches

Note : Bien que MPI 2.0 comporte une instruction `MPI_Spawn`, les communicateurs restent toujours aussi importants!

`MPI_Comm_split( comm, couleur, cle, &newcomm ) :`

- Doit être exécutée par tous les processus
- Crée un ou plusieurs nouveaux communicateurs, en fonction des `couleur`s ayant été spécifiées
- Les processus ayant indiqué la même `couleur` se retrouvent dans le même sous-groupe
- Le numéro du processus dans le nouveau groupe est déterminé par l'ordre des valeurs de `cle` — « *Within a subgroup, processes are ranked according to the value of [this parameter]; ties are broken according to the process' rank in* `comm`.»

Note : Si un processus utilise la couleur `MPI_UNDEFINED`, alors il ne fera partie d'aucun des nouveaux communicateurs

MPI_Comm_split

Creates new communicators based on colors and keys

Synopsis

```
int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *newcomm)
```

Input Parameters

comm communicator (handle)

color control of subset assignment (nonnegative integer). Processes with the same color are in the same new communicator

key control of rank assignment (integer)

Output Parameters

newcomm new communicator (handle)

Notes

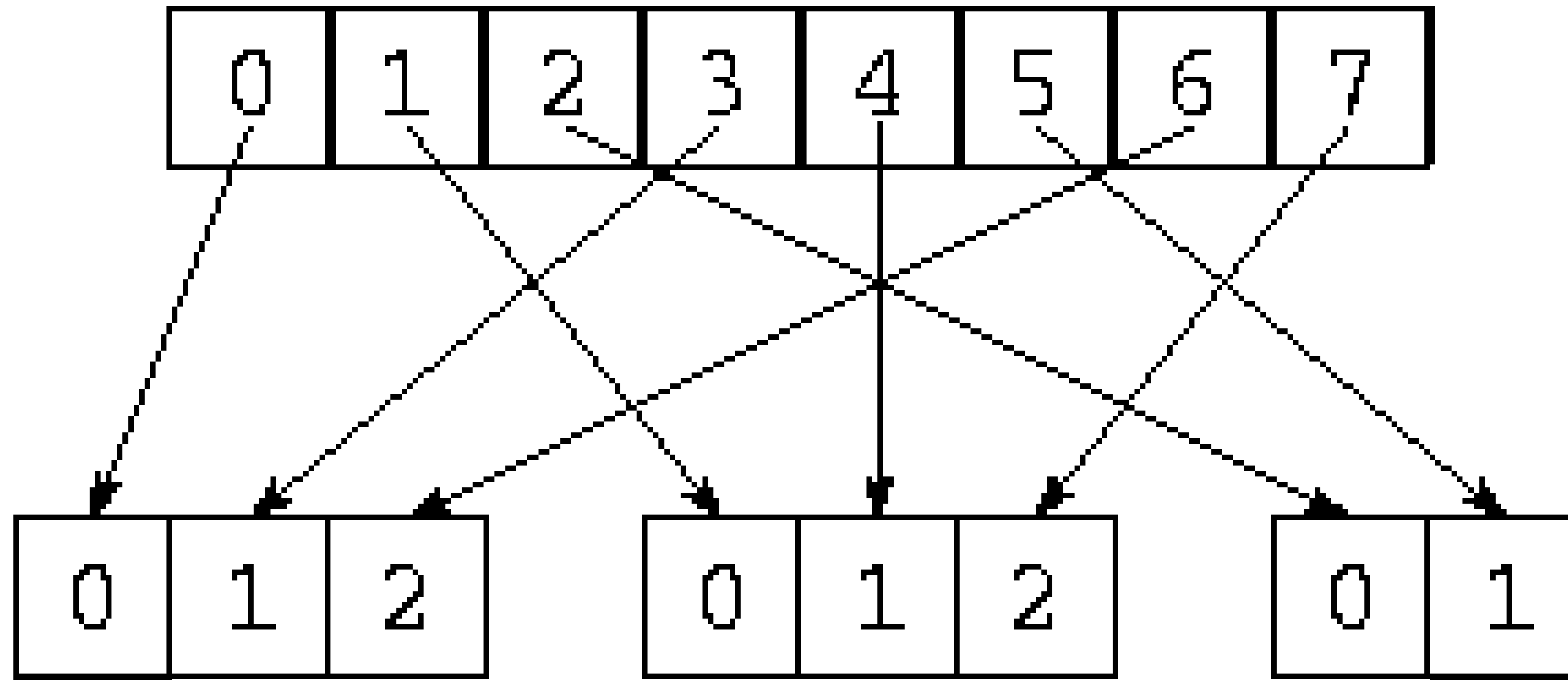
The color must be non-negative or MPI_UNDEFINED.

Algorithm

1. Use MPI_Allgather to get the color and key from each process
2. Count the number of processes with the same color; create a communicator with that many processes. If this process has MPI_UNDEFINED as the color, create a process with a single member.
3. Use key to order the ranks

L'opération suivante crée trois communicateurs, donc trois sous-groupes (Figure 8.10, p. 298 de Foster) :

```
MPI_Comm_rank( comm, &myid )
MPI_Comm_split( comm, myid % 3, myid, &newcomm );
```



Exemple : les deux opérations suivantes ont le même effet, parce que tous les processus spécifient la même `couleur` et `cle`

```
MPI_Comm_split( comm, 0, 0, &newcomm );
```

```
MPI_Comm_dup( comm, &newcomm );
```

Exemple : si un processus ne doit pas faire partie d'un nouveau communicateur, on indique `MPI_UNDEFINED` comme «couleur» :

```
if ( myID == 0 ) {
  MPI_Comm_split( MPI_COMM_WORLD, MPI_UNDEFINED, myId, &newcomm );
} else {
  MPI_Comm_split( MPI_COMM_WORLD, couleur, myId, &newcomm );
}
```

14.6 Autres éléments de MPI

- Types dérivés : nouveaux types définis par le programmeur qui permettent de transférer, dans un seul message, des éléments arbitraires, possiblement non-contigus

  Exemple : l'opération `MPI_Send` suivante transmet les 10 nombres points-flottants `data[0]`, `data[32]`, `data[64]`, etc., et ce en un seul message

```
float data[1024];
MPI_Datatype floattype;

MPI_Type_vector(10, 1, 32, MPI_FLOAT, &floattype);
  // 10: nombre total d'items a transmettre
  //  1: nombre d'items a partir de chaque position
  // 32: intervalle entre les items
MPI_Type_commit(&floattype);
MPI_Send(data, 1, floattype, dest, tag, MPI_COMM_WORLD);
```

- Opérations pour examiner les paramètres de l'environnement d'exécution : `MPI_Get_processor_name`, `MPI_Get_version`, `MPI_Wtick`

- Opérations pour associer diverses informations (paires clés/définitions) aux communicateurs : `MPI_Attr_put`, `MPI_Attr_get`

- Création d'une opérations de réduction spécifiée par l'usager

```
int MPI_Op_create( MPI_User_function *assoc_func,
                   int commutative,
                   MPI_Op *op );
```

- Opérations collectives de dispersion/regroupement avec *nombres variables* d'éléments reçus/transmis pour chaque processus :

```
int MPI_Scatterv( void *sendbuf,
                  int *sendcnts, // Tableau de taille = nb. de procs. de comm
                  int *displs,   // Tableau de taille = nb. de procs. de comm
                  MPI_Datatype sendtype,
                  void *recvbuf,
                  int recvcnt,
                  MPI_Datatype recvtype,
                  int root,
                  MPI_Comm comm );

int MPI_Gatherv( void *sendbuf,
                 int sendcnt,
                 MPI_Datatype sendtype,
                 void *recvbuf,
                 int *recvcnts, // Idem
                 int *displs,   // Idem
                 MPI_Datatype recvtype,
                 int root,
                 MPI_Comm comm );
```

- Opération d'échange point-à-point combinant **envoi et réception** :

```
int MPI_Sendrecv( void *sendbuf,
                  int sendcount,
                  MPI_Datatype sendtype,
                  int dest,
                  int sendtag,

                  void *recvbuf,
                  int recvcount,
                  MPI_Datatype recvtype,
                  int source,
                  int recvtag,

                  MPI_Comm comm,
                  MPI_Status *status );
```

- Autres opérations de communication collective :
  - `MPI_Allgather`, `MPI_Allgatherv`
  - `MPI_Alltoall`, `MPI_Alltoallv`
  - etc.

- Autres modes d’envoi :

  MPI has a number of different "send modes." These represent different choices of buffering (where is the data kept until it is received) and synchronization (when does a send complete). In the following, I use "send buffer" for the user-provided buffer to send.

  - `MPI_Send`

    `MPI_Send` *will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive.*

  - `MPI_Bsend`

    May buffer; returns immediately and you can use the send buffer. A late add-on to the MPI specification. Should be used only when absolutely necessary.

  - `MPI_Ssend`

    Will not return until matching receive posted

  - `MPI_Rsend`

    May be used ONLY if matching receive already posted. User responsible for writing a correct program.

– `MPI_Isend`*:*

Nonblocking send. But not necessarily asynchronous. You can NOT reuse the send buffer until either a successful, wait/test or you KNOW that the message has been received (see `MPI_Request_free`*). Note also that while the I refers to immediate, there is no performance requirement on* `MPI_Isend`*. An immediate send must return to the user without requiring a matching receive at the destination. An implementation is free to send the data to the destination before returning, as long as the send call does not block waiting for a matching receive. Different strategies of when to send the data offer different performance advantages and disadvantages that will depend on the application.*

– `MPI_Ibsend`

buffered nonblocking

– `MPI_Issend`

Synchronous nonblocking. Note that a Wait/Test will complete only when the matching receive is posted.

– `MPI_Irsend` *As with* `MPI_Rsend`*, but nonblocking.*

Note that "nonblocking" refers ONLY to whether the data buffer is available for reuse after the call. No part of the MPI specification, for example, mandates concurent operation of data transfers and computation.

Source : `http://www.mcs.anl.gov/research/projects/mpi/sendmode.html`

14.7 Autres versions de MPI

14.7.1 MPI-2 = Version 1997

- Création dynamique de processus :

```
int MPI_Spawn( char *program, char *argv[], int maxprocs,
               MPI_Info info, int root,
               MPI_Comm parents, MPI_Comm *children, int errors[] );
```

- Communication unidirectionnelle ($1^{ère}$ version) : `MPI_Put`, `MPI_Get`
- *Message handlers*

```
int MPI_Errhandler_set( MPI_Comm comm, MPI_Errhandler errhandler );
```

- Interface C++

En tout : 120 nouvelles fonctions ($\Rightarrow$ total de 249 fonctions!)

14.7.2 MPI-3 = «Nouvelle» version (2010 ☹, 2011 ☹, 2012 ☺ !)

- Opérations collectives de communication **non-bloquantes** :

```
MPI_Request req;
MPI_Status stat;

MPI_Ibcast( &buf, 1, MPI_INT, 0, comm, req )
...
MPI_Wait( &req, &stat );
```

- Tolérance aux pannes
- Accès mémoire à distance (*Remote memory access*)
- Meilleure interface pour l'utilisation de. . .
  - *threads* (Pthreads, OpenMP)
  - GPU (OpenCL, CUDA)

14.8 Conclusion : Pourquoi MPI est si gros!?

> *One aspect of concern, particularly to novices, is the large number of routines comprising the MPI specification. In all there are 128 MPI routines, and further extensions will probably increase their number. There are two fundamental reasons for the size of MPI. The first reason is that MPI was* **designed to be rich in functionality**. *This is reflected in MPI's support for derived datatypes, modular communication via the communicator abstraction, caching, application topologies, and the fully-featured set of collective communication routines. The second reason for the size of MPI reflects the* **diversity and complexity of today's high performance computers**. *This is particularly true with respect to the point-to-point communication routines where the different communication modes arise mainly as a means of providing a set of the most widely-used communication protocols. [...] The availability of a large number of calls to deal with more esoteric features of MPI allows one to provide a simpler interface to the more frequently used functions.*

Source : `http://www.netlib.org/utk/papers/mpi-book/node198.html`

Et ce texte parlait alors de MPI 1.0, avec une centaine de constantes et fonctions, et pas de MPI 3.2, avec près de 400 constantes et fonctions (sic)!